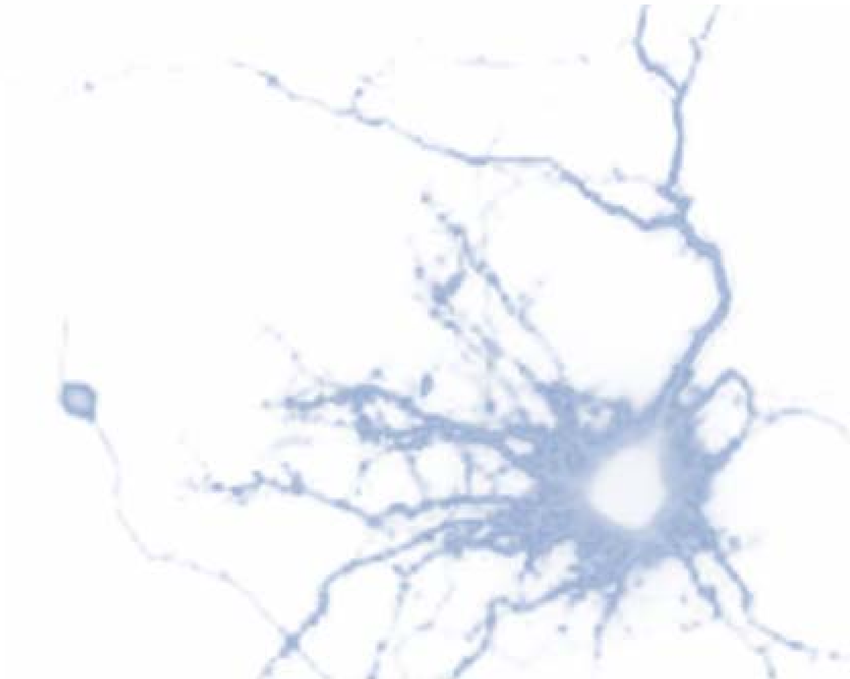
ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Cellular Architectures Research Group (CARG)

A MOVE PROCESSOR FOR BIO-INSPIRED SYSTEMS

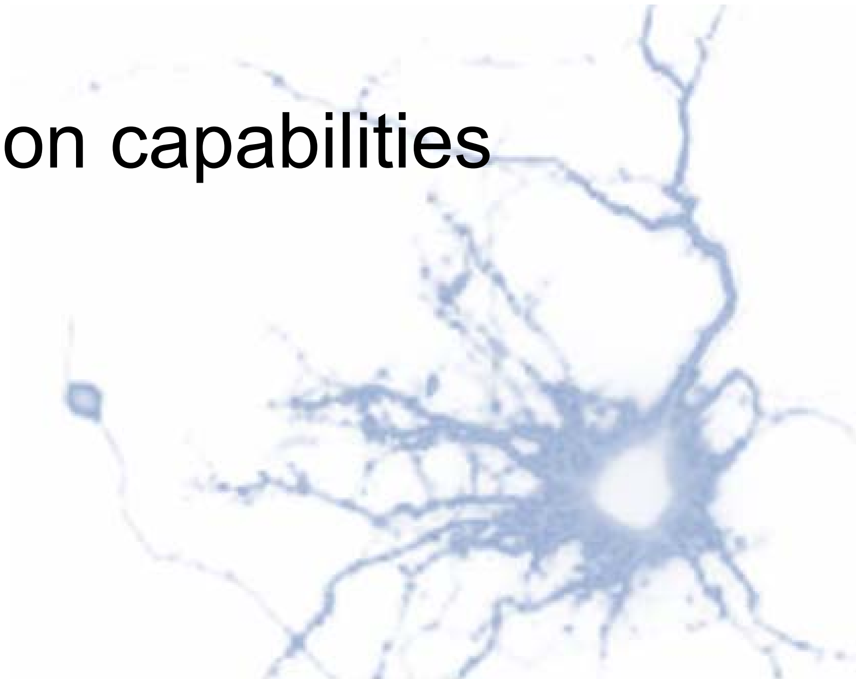


Pierre-André Mudry, Prof. Gianluca Tempesti

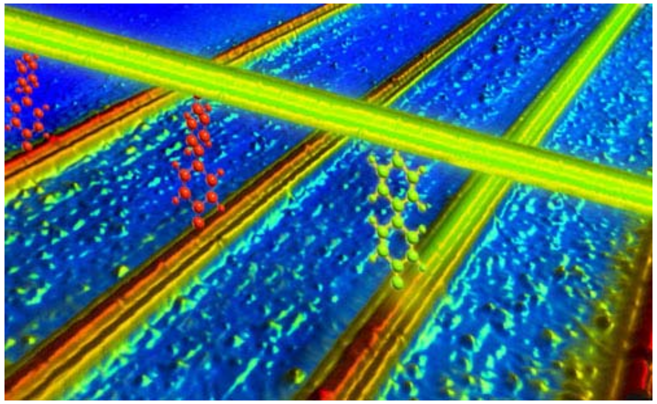
NASA/DoD Conference, June 2005

Biological systems inspiration

- Very complex, large scale systems
- Self-assembling according to a small genome
- Self-repairing
- Have learning and evolution capabilities
- Extensive parallelism



Motivations

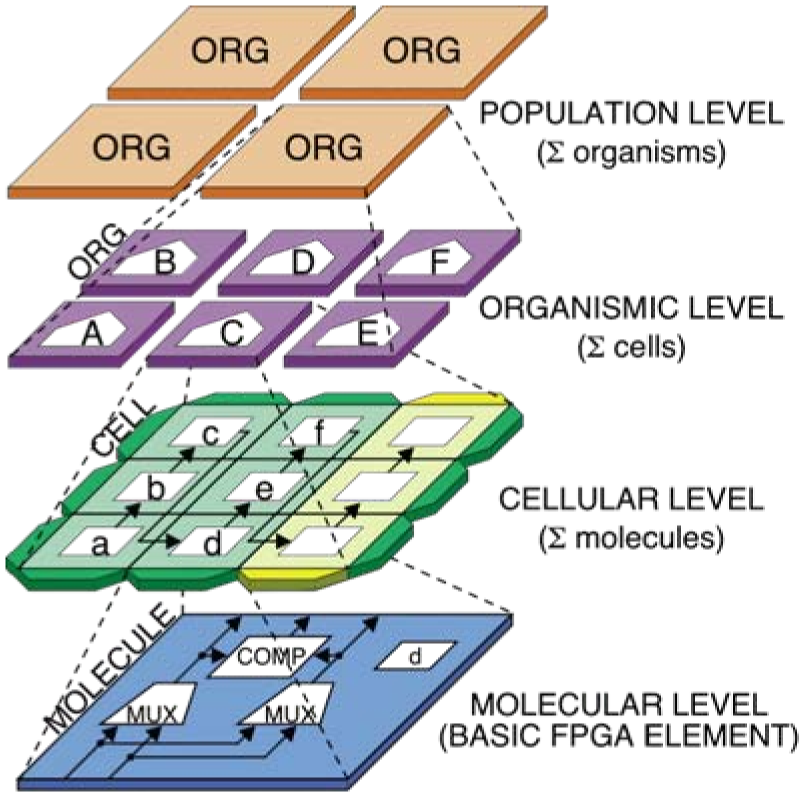


- Nanoelectronics promises:
  - Density increase
  - Size shrink
  - Molecular electronics
- New challenges arise
  - Design time / cost
  - Synthesis issues
  - System verification

How to cope with all this new hardware ?

Ontogenetic processor architectures

- Ontogenesis
  - Molecules self-assembly into cells
  - Cells self-assembly into organisms
  - According to a very compact set of instructions
- How can it be used ?
  - Cell = processor
  - Organism = massively parallel multiprocessor system

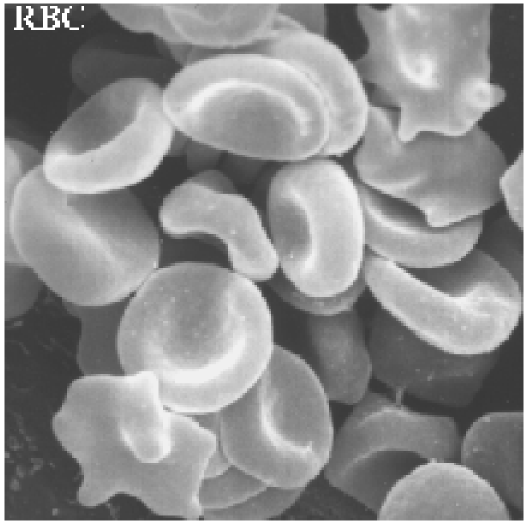


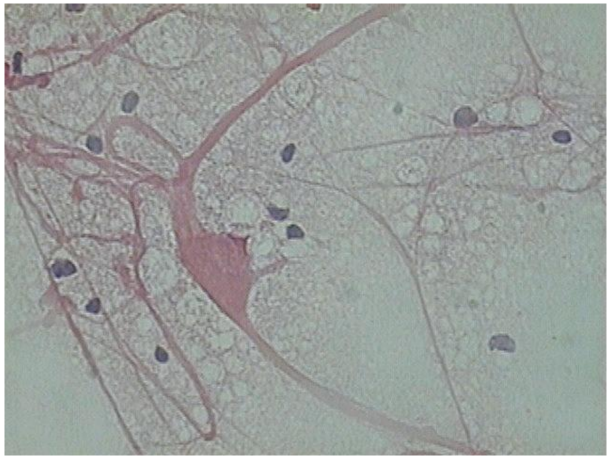
Ontogenetic processor systems specifications

- Cells are *application-specific processors*
  - Instruction-set targeted for a single application
  - Simpler and more compact programs
- Communication among cells made with an adaptive network → data flow computation
- Can lead to *fault-tolerant* processor arrays through cell replacement (cicatrisation)

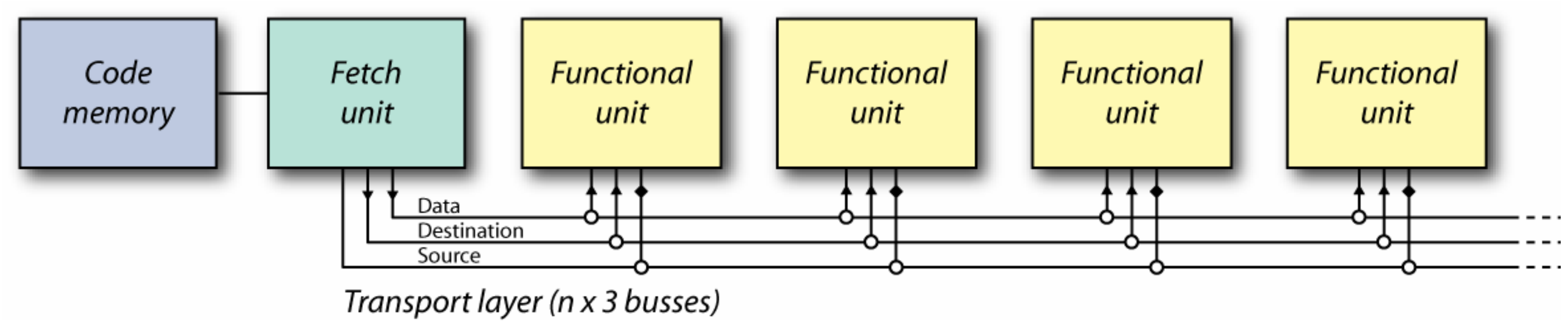
Why not a standard processor ?

- Biological cells are structurally adapted to their functions (which simplifies their operation)
- In standard processor architectures, the structure can not be deeply modified without drastic repercussions at many levels (instruction decoding, assembly language)



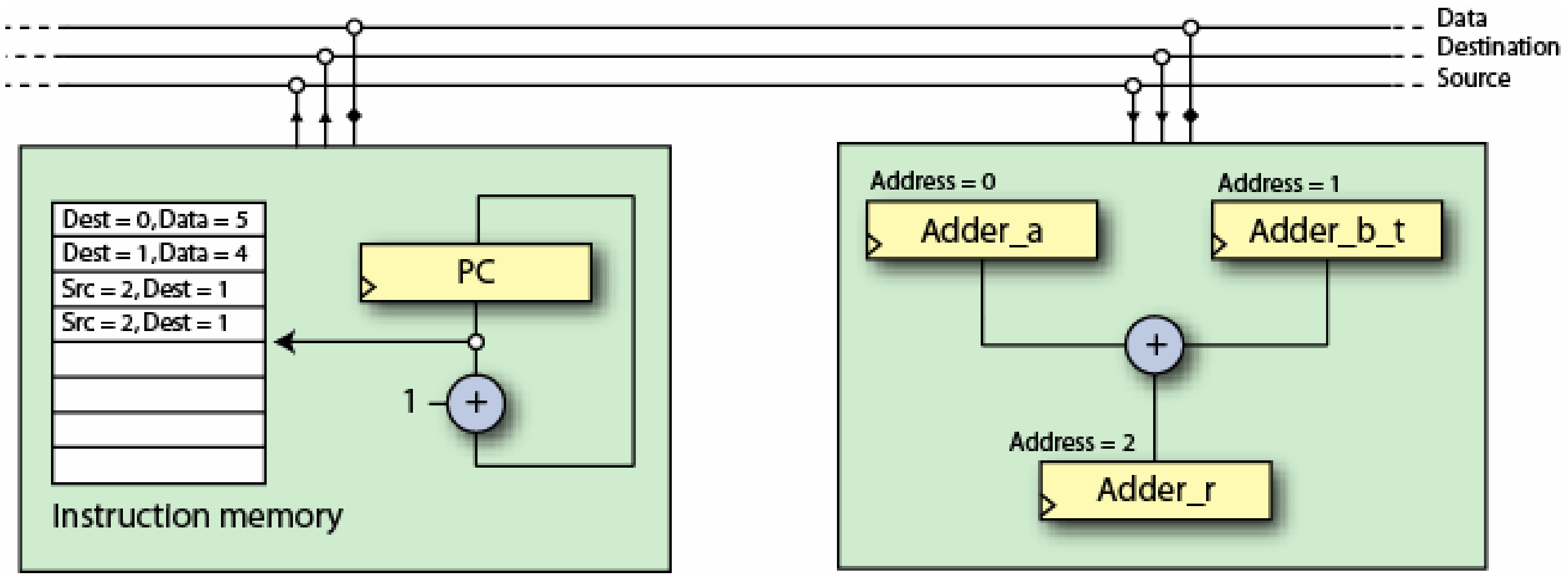


The MOVE processor architecture

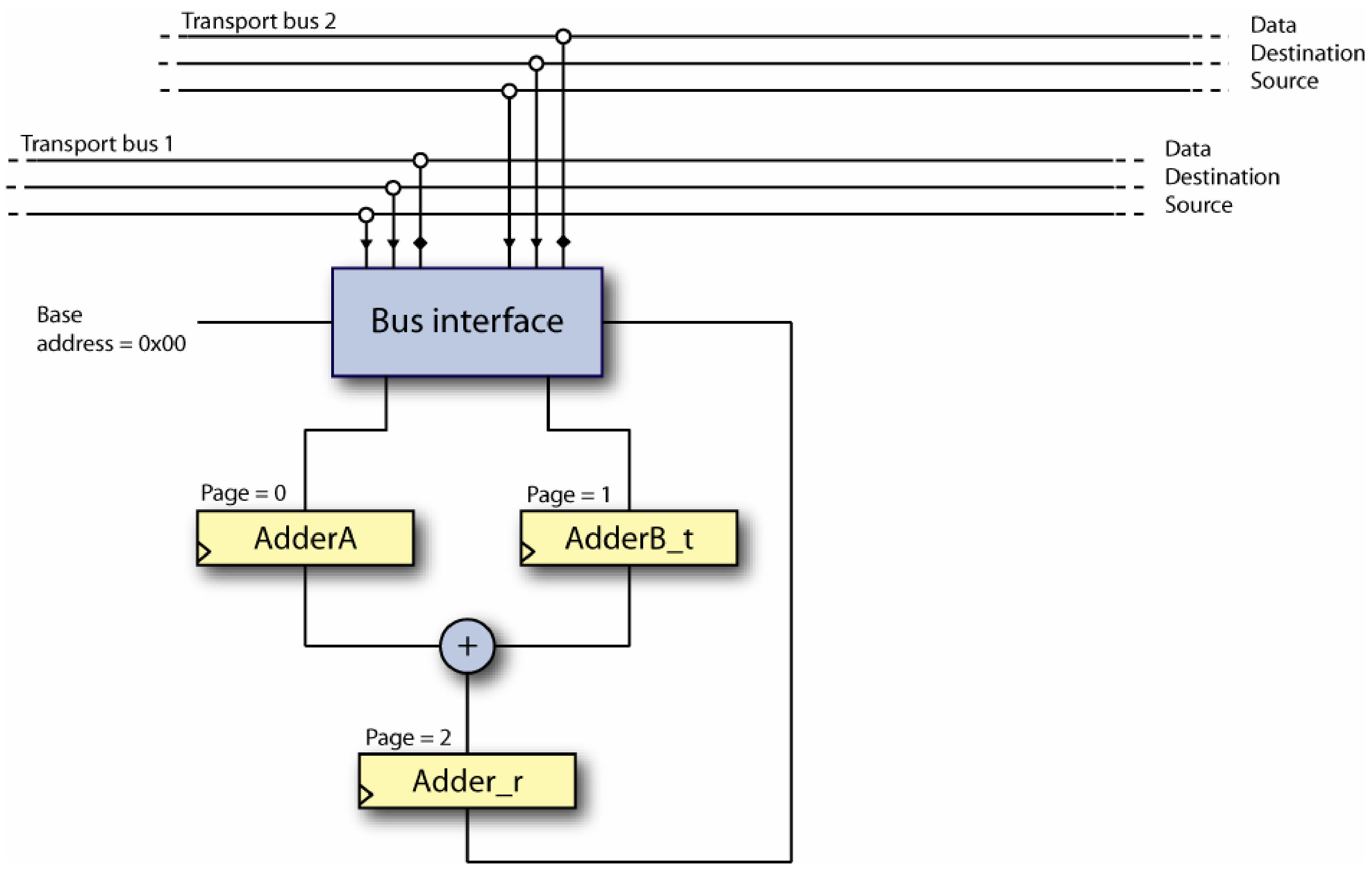


- One instruction : *move*
- Data displacements trigger operations
- Architecture based around data ≠ operation centric
- Regular structure : functional units + data network
- Scalable and modular architecture (generic interface)

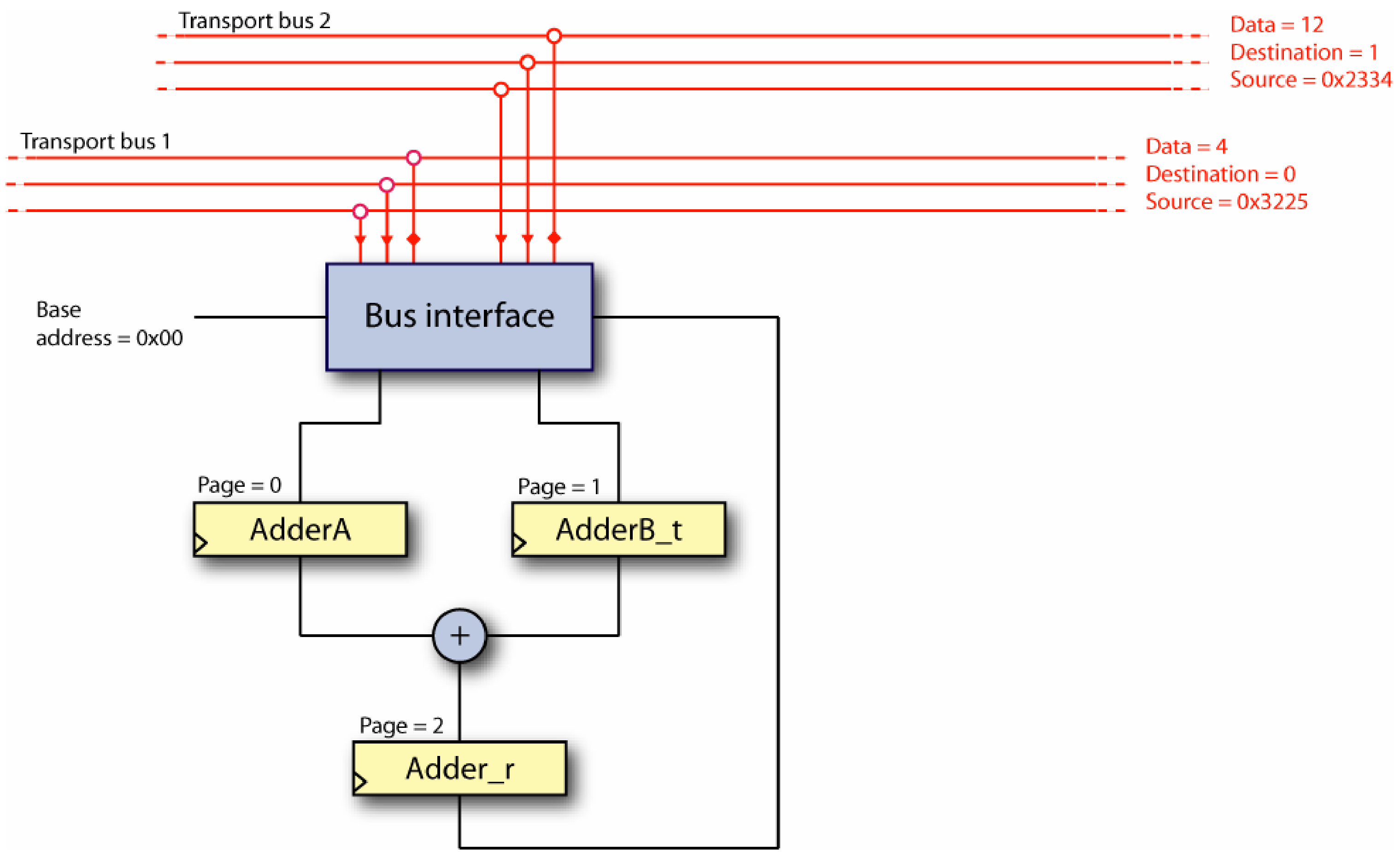
Operation principle



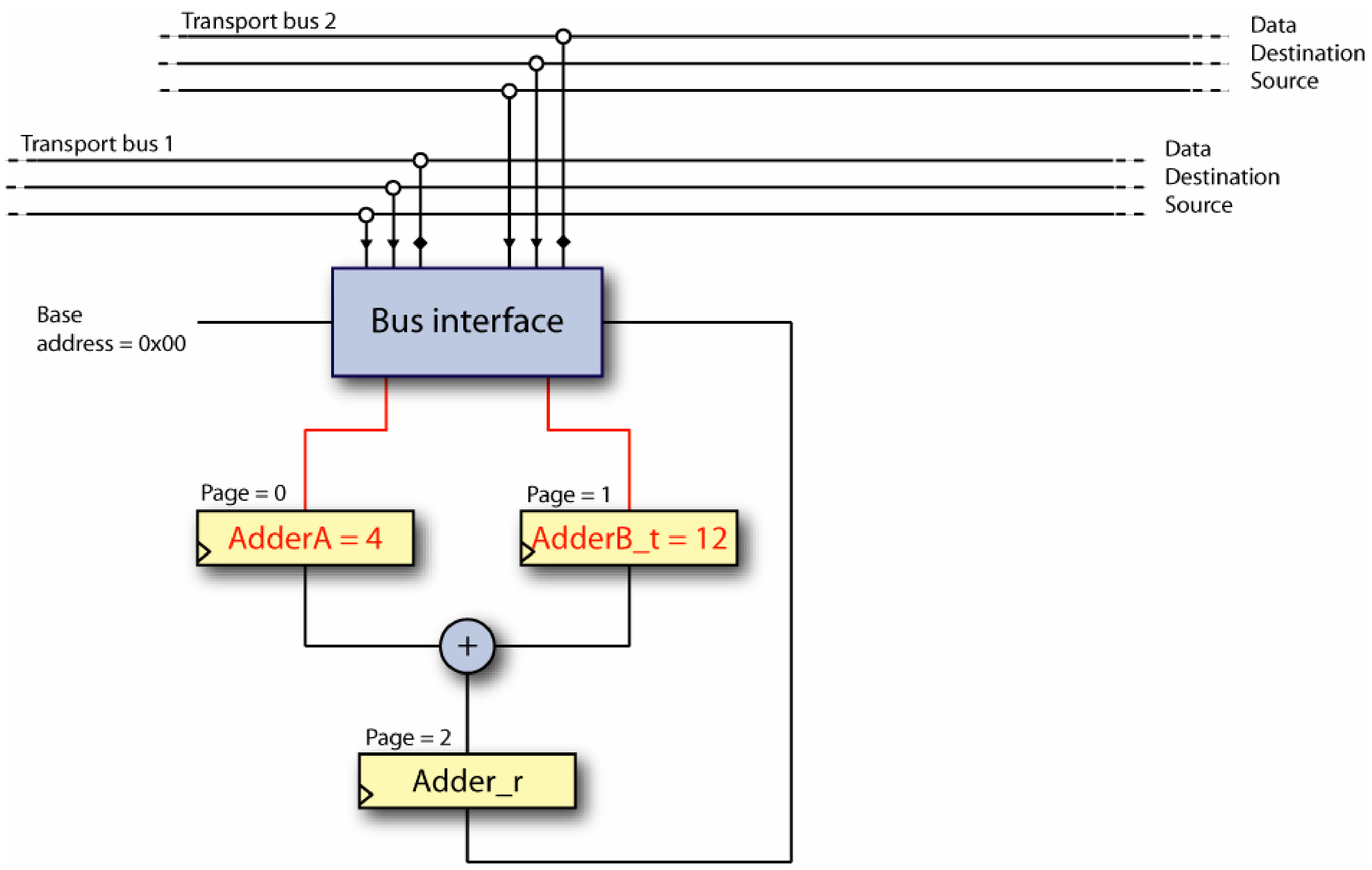
« *Add* » equivalent example



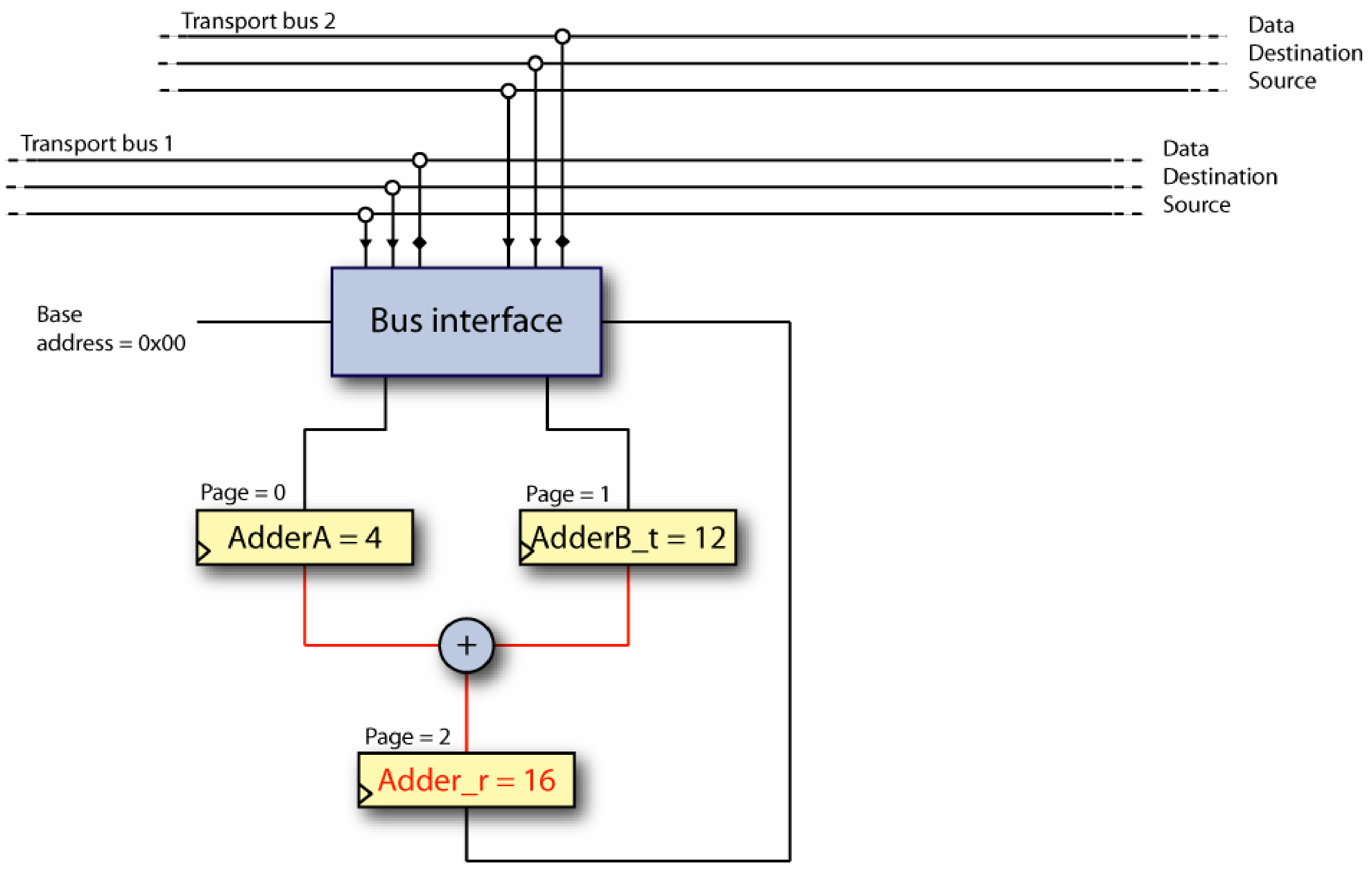
« *Add* » equivalent example



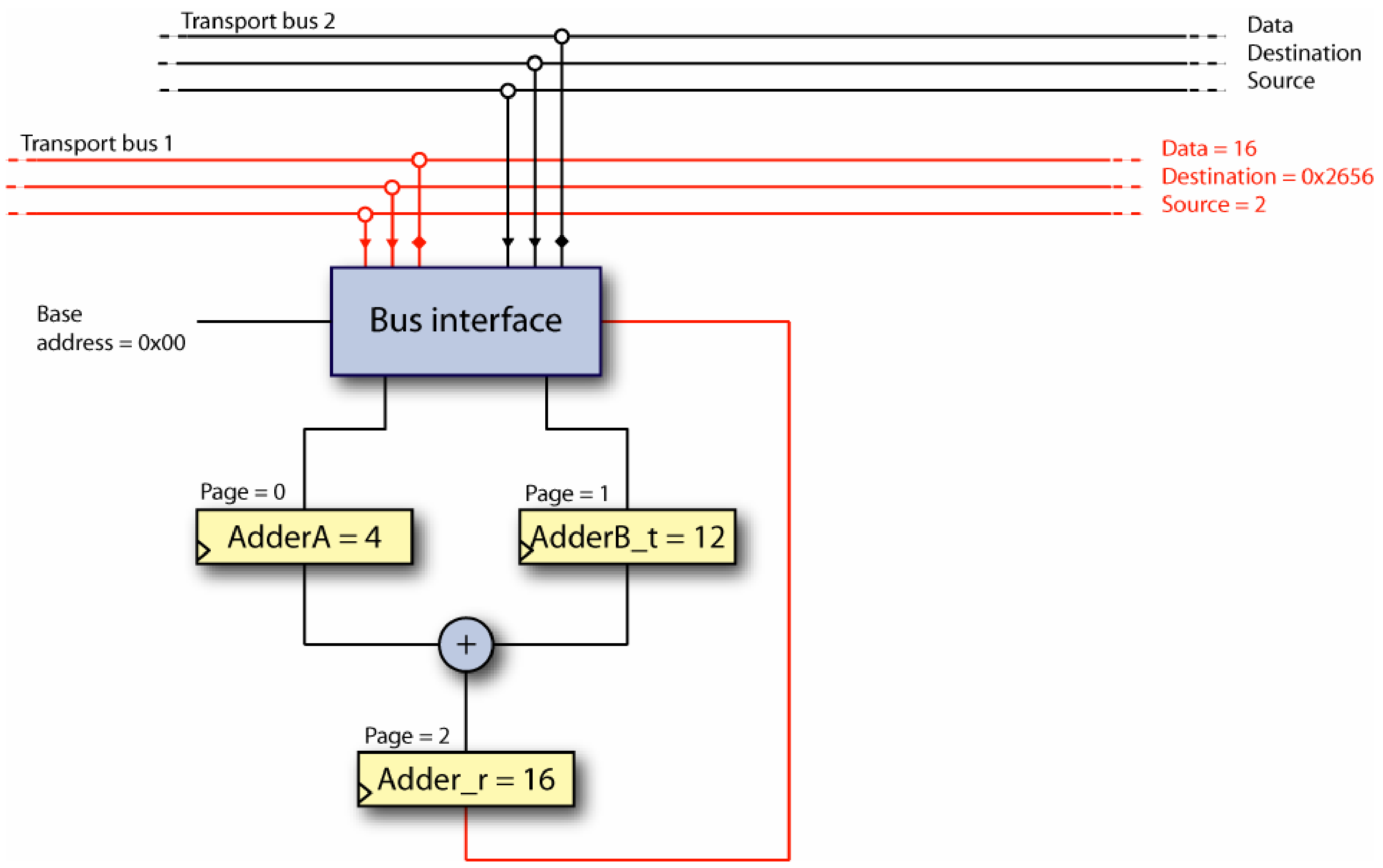
« *Add* » equivalent example



« *Add* » equivalent example



« *Add* » equivalent example



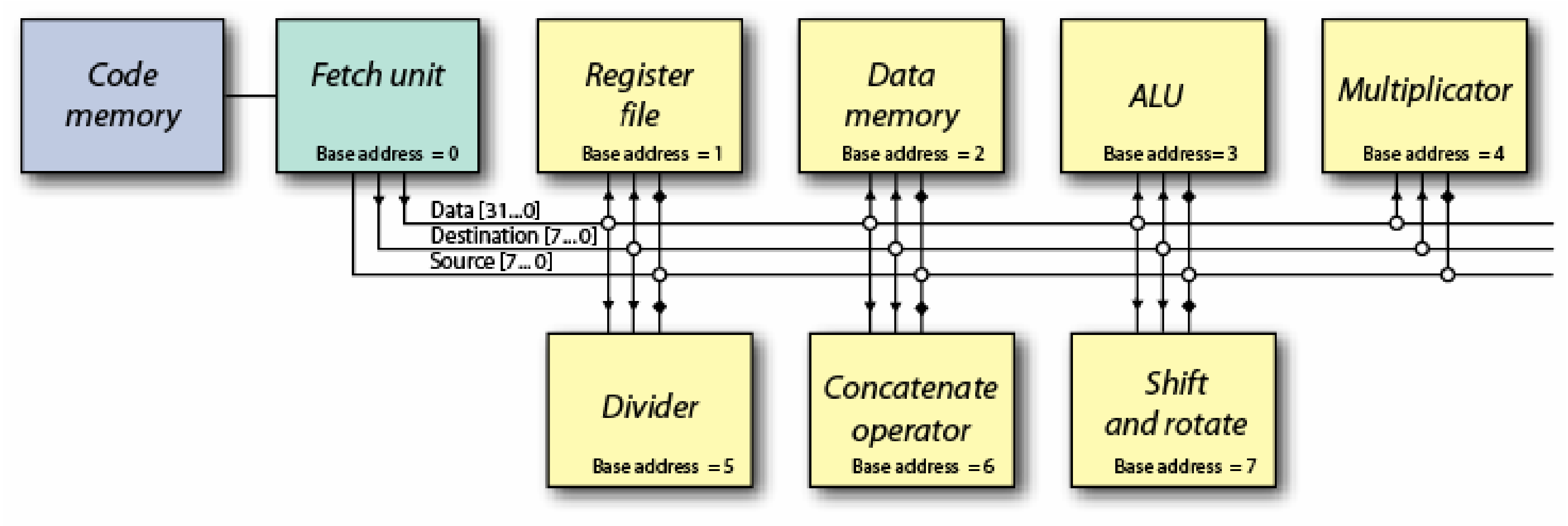
MOVE architecture

- The pros :
  - Reduce register file transfers
  - Simple decode logic (one instruction)
  - Instruction-level parallelism through VLIW
  - Flexibility of instruction scheduling
  - **Modularity** → all units are replaceable and are seen as black-boxes
- The cons :
  - Difficult to program at low level
  - Program size can be bigger if no application-specific FUs are used
  - Not the most efficient…

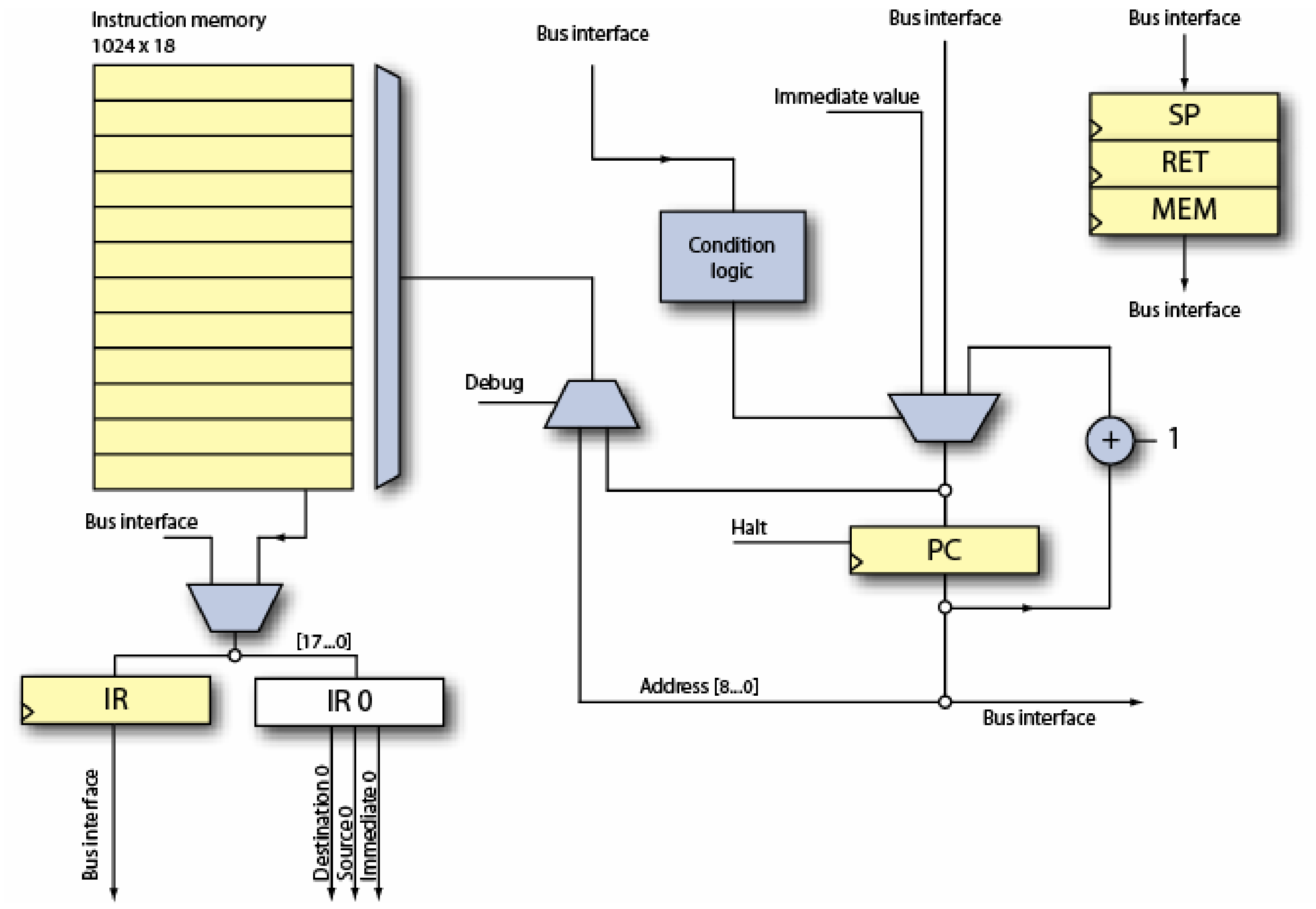
The Ulysse processor

- VHDL soft-core
- Can be implemented using memory embedded in a FPGA
- Fully configurable
  - VLIW
  - Datapath width
  - FU choice
  - I/O and communication units are standard FUs

Internal architecture



The fetch unit

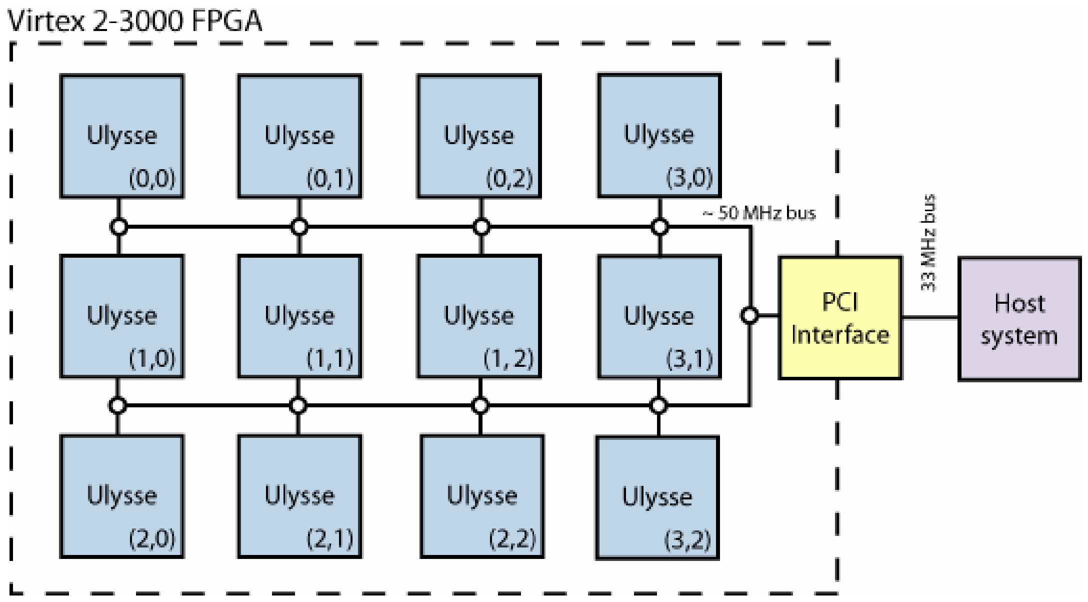


Genericity and scalability

```
entity ulysse_main is
    generic
    (
      -- Functional units  instanciation
      mult_present : natural range 0 to 1 := 0;
      dct_present : natural range 0 to 1  := 0;
      divider_present : natural range 0 to 1 := 0;
      fifo_present : natural range 0 to 1 := 0;
      [...]
    );
```

- Unit instanciation (expression power)
- FU library
- Application-specific processor generation

Our prototype



- 12 processors on a single chip, running at 50 MHz
- Communication network = shared bus
- Connected to a PC through PCI
- DMA transfers
- Regular structure of identical processors (ontogenesis)

The pros…

- The processing power can easily be chosen according to a given budget (even for small FPGAs)
- Code and hardware **reuse**
  - NRE reduction (once the libraries are available)
  - Decode and memory logic are “almost” constant
- Good trade-off between ASIP and general purpose approach
- Network synthesis can cope with arbitrarily complex connection schemes (communication units are functional units)

The cons…

- Less powerful than dedicated hardware

  → price of the compromise
- Reconfiguration requires a new synthesis phase

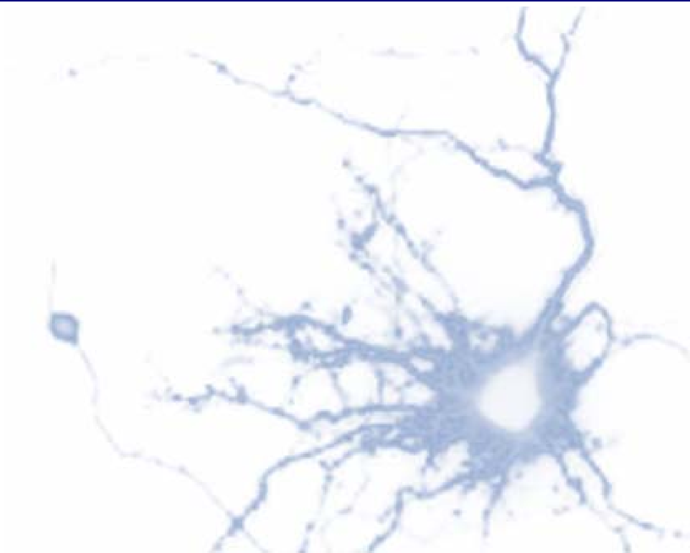
  → partial, dynamic reconfiguration
- Very complicated to program
  - MOVE code is difficult to write and optimise
  - Hardware and software have to be co-designed

  → the software layer has to be developed

The opportunities…

- Might be a possible answer for large scale parallel circuit synthesis (copy of identical structures through self-replication)
- The complexity of the design process (co-design) can open opportunities for evolutionary algorithms (evolution of the circuit can be made at the FU level)
- Arbitrarily complex communication patterns simplify the design of complex neural network architectures
- FU modularity
  - Simplifies the differentiation process
  - Allows automatic processor generation / assembly

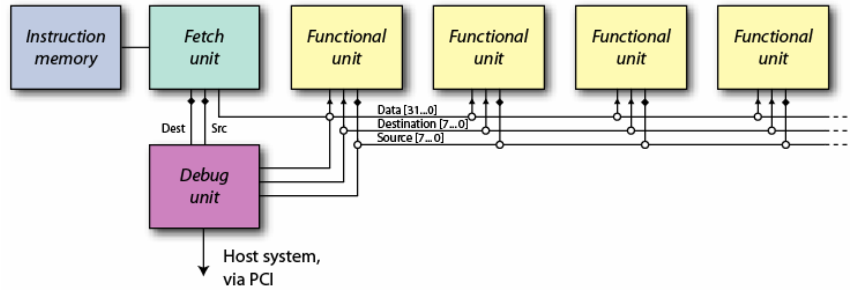
Thank you for your attention

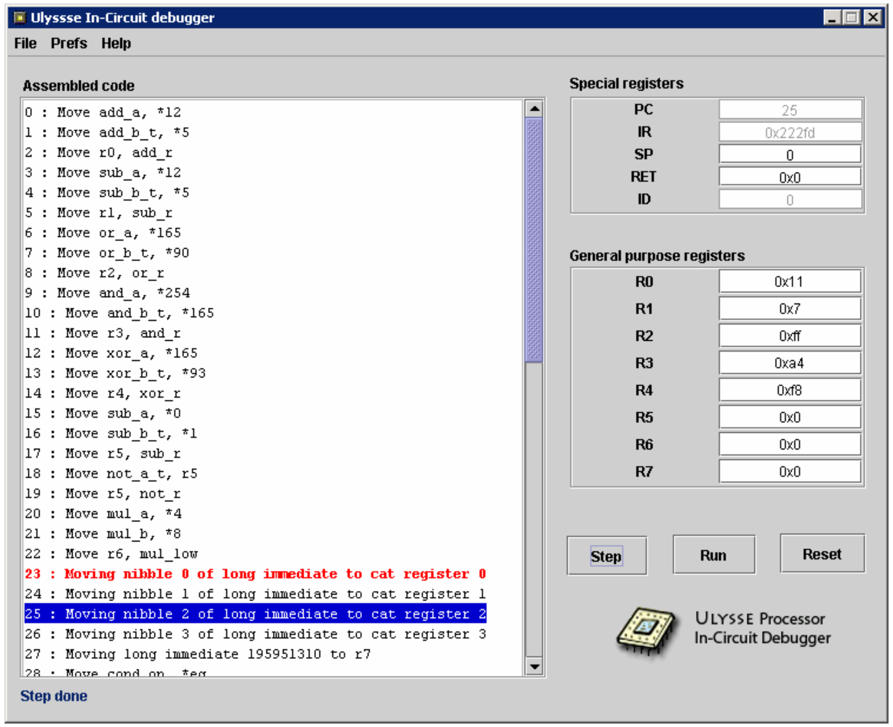


Pierre-André Mudry, CARG, EPFL

Main features (I)

- Debug mode (external control of the bus for monitoring), could be used for ontogenetic mechanisms
- In-circuit debugger
- Test suite (ModelSim interaction)





Ulysse In-Circuit debugger
File Prefs Help
Step
Run
Reset
ULYSSE Processor In-Circuit Debugger

Assembled code

```
0 : Move add_a, *12
1 : Move add_b_t, *5
2 : Move r0, add_r
3 : Move sub_a, *12
4 : Move sub_b_t, *5
5 : Move r1, sub_r
6 : Move or_a, *165
7 : Move or_b_t, *90
8 : Move r2, or_r
9 : Move and_a, *254
10 : Move and_b_t, *165
11 : Move r3, and_r
12 : Move xor_a, *165
13 : Move xor_b_t, *93
14 : Move r4, xor_r
15 : Move sub_a, *0
16 : Move sub_b_t, *1
17 : Move r5, sub_r
18 : Move not_a_t, r5
19 : Move r5, not_r
20 : Move mul_a, *4
21 : Move mul_b, *8
22 : Move r6, mul_low
23 : Moving nibble 0 of long immediate to cat register 0
24 : Moving nibble 1 of long immediate to cat register 1
25 : Moving nibble 2 of long immediate to cat register 2
26 : Moving nibble 3 of long immediate to cat register 3
27 : Moving long immediate 195951310 to r7
28 : Move cond_op, *eq
```

Special registers

PC	25
IR	0x222fd
SP	0
RET	0x0
ID	0

General purpose registers

R0	0x11
R1	0x7
R2	0xff
R3	0xa4
R4	0xf8
R5	0x0
R6	0x0
R7	0x0

Step done

Main features (II)

Without macro-assembly

```
 begin:
    move jump_condition, #1;
    move jabs_cond, #after ; // Shall normally jump

    move assert_line, #current_line;
    move assert_a_t, #1 || move assert_b, #2; // Hangs if here

after:
    move adder_b_t, #1;
    nop;
    move r0, adder_r;
    move cond_op, #zero || move cond_a_t, r0;
    nop;
    move jump_condition, cond_result;
    move jabs_cond, #dead_end;
    move assert_stop, #0;

dead_end:
    move assert_line, #current_line;
    move assert_a_t, #1 || move assert_b, #2; // Hangs if here
```

With

```
#include “macros.inc”

begin:
  jump after; // Shall normally jump
  error; // Hangs if here

after:
  add #0, #1;
  jz add_result, dead_end;
  exit; // Everyhing’s ok

dead_end:
  error;
```

Only useful for assembly programming…